ELASTIC BUILD SERVICE

Ville Seppänen
Tieto Corporation / Tampere University of Technology
P.O. Box 527, 33101 Tampere, Finland
ville.t.seppanen@tut.fi

Abstract

Linux-based operating systems such as MeeGo consist of thousands of modular software packages. Compiling and packaging source code is an automated but computationally heavy task. As the load on a build farm can vary greatly, a local infrastructure is difficult to provision efficiently. In this paper we present the elastic acquisition of cloud resources as a means to ensure sufficient computing capacity for a software build system. This system is Open Build Service, a centrally managed distributed build system capable of building packages for several distributions and architectures. Main concerns were the technical feasibility, security and cost-efficiency of the proposed solution. A script was implemented to autonomously manage the elastic cloudbursting, monitoring resource usage and demand and making decisions whether new machines should be requested or idle ones terminated. The latencies incurred by the physical distance to the cloud were not insurmountable and the system scaled up in a matter of minutes. The main advantage achieved with cloud usage in this work was the advent of seemingly infinite number of resources on-demand, ideal for building taking care of sudden bursts of packages that can be built in parallel.

Keywords: Cloud computing, cloudbursting, build system, software build, OBS, Open Build Service, computing cluster, distributed system, virtualization

1 Introduction

Operating systems have grown in complexity over the last ten years. Modern Linux-based operating systems consist of thousands of modular software packages. These packages are created with automated build tools which require a lot of computational power. Furthermore, the building needs of software developers may vary greatly, causing irregular load spikes on the build system. This makes it difficult to successfully provision a dedicated infrastructure; overprovisioning the build cluster to cope with even the largest temporary spikes would be costly, while underprovisioning would lengthen build times during load spikes.

Cloud services, Infrastructure-as-a-Service more specifically, provide easily obtainable, utility-like computing power over the Internet in the form of virtual machines. Users can request resources from the cloud as necessary, usually over an Application Programming Interface (API) or a web-based user interface, and pay based on the actual usage.

In this paper we present the elastic acquisition of cloud resources as a means to ensure sufficient computing capacity for a software build system. This system is Open Build Service, a centrally managed distributed build system capable of building packages for MeeGo among other distributions. Main concerns were the technical feasibility, security and cost-efficiency of the proposed solution.
2 Cloudbursting – Elastic hybrid clouds

The key motivations for cloud customers are the economic benefits from achieved elasticity and transference of risk in regards of provisioning. Cloud computing can be seen as transforming capital expenditures into operational expenditures. Maintaining an infrastructure that can sustain even the highest peaks can be costly, especially if the peaks are rare and far larger than the average load [1].

Multiple public or private clouds can be used in conjunction, forming a hybrid cloud. The underlying use of several clouds is often hidden from the end-user, but provisioning should take into account the heterogeneity of the providers: their service level agreement, billing and current state. Because of this, hybrid clouds can be more difficult to manage effectively.

Cloudbursting is a concept of expanding a pool of local resources into a public cloud when local capacity reaches its limit. It essentially creates a scalable and elastic hybrid cloud infrastructure, on-demand without manual intervention [1]. This is especially efficient in handling load bursts; allowing extra jobs to overflow to the cloud and making sure applications remain available when local resources become saturated. Marshall et. al [2] have presented a model of an elastic site for describing cloudbursting, shown in Figure 1.

![Figure 1. Elastic site model](image)

In this model, the resource usage of a static, local site is monitored by a manager component and additional resources are requested from the cloud based on the demand. This forms an elastic site, capable of adapting to fluctuating resource demand. A similar resource manager is implemented in this work, termed as a service manager in accordance with RESERVOIR’s reference architecture of cloud management [3].
3 Software building with Open Build Service

Software building is a process where software is compiled from source code and bundled with configuration data into packages. Packages are very prominent in Linux-based operating systems, where applications and the operating system itself are composed of packages. Packages are used to ease installation, updating, and uninstalling of software. In addition to the actual code, a package holds metadata such as package name, version, how to install and uninstall it and what dependencies it has. To lessen the amount of redundant code, packages can depend on other packages, forming a dependency tree. This metadata originates from the recipe (In RPM, this file is called a spec file) of the package, a file that holds the metadata and instructions on how to build the package.

Open Build Service (OBS), is an open-source, cross-distribution development platform [4], which is used to build software packages for several Linux distributions (e.g. MeeGo and openSUSE) and for different architectures (e.g. ARMv7 and x86). Software developers write source code and package descriptions and the system takes care of package dependency changes and rebuilds packages automatically. Due to the modularity of packages, they can be built in parallel. OBS is a distributed system, where a head node manages the system and dispatches build jobs to multiple worker nodes. The head node composes from several software services, which can be spread to separate hardware [5], a moderately scaled out example is shown in Figure 2.

**Figure 2.** Example structure of Open Build Service
The front-end consists of a web server and an XML API service, which is used by the command line client (OSC). The storage node has source code repositories and a database for persistent data. When users make changes to the source code, the source code service notifies the back-end. The back-end hosts most of the decision-making components of OBS. When the job scheduler gets notified of source code changes, it calculates build dependencies and generates jobs into a queue for the dispatcher. The dispatcher assigns available jobs to idle workers. When a job is completed, the resulting package is signed as authentic and published.

Each worker can build a single package at a time, and by default the number of workers on a build host equals to the number of its processor cores. When a worker is assigned a build job, it downloads prerequisite packages from the repository and sets up a build environment. This has to be recreated for each build to ensure reproducibility. The worker downloads source files and compiles and packages the code. The resulting package is then sent to the package repository. Depending on the hardware, a build job can take from a minute for a single small package to hours or days for building a complete operating system.

4 Extending MeeGo building to the cloud

The overall requirement for the build system was that it would automatically acquire cloud resources when local infrastructure becomes saturated. This requires active monitoring of both work queue and current build jobs and allocating more resources only when really needed. Based on the elastic site model and the cloud management reference architecture, several components were picked to form a proposed architecture for an elastic build service, presented in Figure 3.

![Proposed architecture for an elastic build service](image-url)
Due to time constraints, the implemented architecture was simplified. Local Build Hosts (LBH) were created on static hardware and managed manually, while Cloud Build Hosts (CBH) were launched only in the Elastic Compute Cloud (EC2) of Amazon Web Services (AWS). Monitoring and management of CBHs were also done using AWS.

The local infrastructure was set up on two separate computers, one working as the OBS server and the other as an LBH. The service was created from the latest stable OBS appliance image loosely following the OBS and MeeGo community-made instructions. Once the build service was working locally, the cloud extension was added.

Each virtual machine in EC2 is created from an Amazon Machine Image (AMI). The CBHs would be launched from a generic CBH AMI and then customized during boot-time using the User Data mechanism of EC2. To do this, a virtual machine was launched from a public openSUSE AMI, configured to run OBS worker and connect to the OBS server through a VPN tunnel, and bundled back into a new AMI. Virtual machines manually launched from this AMI would boot up, connect to the OBS server and start building packages, becoming CBHs.

A controller for automating the elasticity was still needed. OBS itself handles scheduling and dispatching centrally, but is not capable of spawning new workers, let alone new virtual machines. The autoscaling feature would need to communicate tightly with the OBS server, thus a simple service manager script was needed. It was implemented in Python as both AWS and OBS have feature-rich Python modules, open-source boto and OSC respectively. Connections between these modules are shown in Figure 4.

![Diagram of script components and their connections](image)

**Figure 4.** Script components and their connections

The service manager was designed so that it could be ran manually or scheduled with cron. When it is executed, it gathers all the necessary data it needs to understand the current situation and makes decisions on whether it should launch or terminate CBHs or do nothing. The manager script has been made publicly available under GPL [6].

First, the service manager connects to AWS management and monitoring servers and requests metadata of existing CBHs. This data includes virtual machine ID, name tag, state, launch
timestamp, IP address and the latest measured CPU utilization. Second, the service manager connects to the local OBS server and requests metadata of both the current workers (hostname, state, packages being built if any) as well as the queue of future jobs (number of jobs waiting and number of blocked jobs).

As virtual machine can only be referenced in EC2 by their virtual machine ID (generated by EC2 itself), and as workers running on the virtual machines are only identified in OBS by their hostname, these two datasets need to be mapped to each other. Before requesting a new CBH, the service manager generates a unique name using the UNIX timestamp (e.g. cbh-08808583). When requesting a new CBH, this name is attached to the virtual machine as an EC2 tag, a key-value pair which allows arbitrary metadata to be stored. In addition, a short shell script is passed to the machine via EC2 User Data which changes the hostname of the machine to be the same as in the tag. This way, a CBH that is idle according to the OBS API can be terminated in EC2.

After collecting data of the current situation, the service manager goes through all the found CBHs to be nominated for termination. Build hosts that are not connected to the OBS server or that are idle even though there is work to be done will be nominated. However, as it takes a varying amount of time for the instance to start up, create a VPN connection and connect to the OBS server (usually 3 minutes), newborn build hosts will be ignored until they reach a boot time threshold (set to 5 minutes).

Idle hosts that have no more work and that are close to a new payment are also terminated. Virtual machines in EC2 are each billed hourly, for every full hour counting from the time the machine was started. Starting a new machine and letting it run for ten minutes and repeating it five more times in one hour, will cost six instance-hours instead of one. Starting and stopping instances should therefore be minimized. Idle build hosts will be kept waiting for new jobs until their lifetime approaches another full hour.

When bad and unnecessary build hosts have been terminated, the service manager determines whether it should request new build hosts. Each machine that is considered still booting up is reserved to have a job from the queue. Subtracting the number of booting workers from the number of jobs in the queue equals to the number of how many more workers are needed. The manager also has limits for the total number of build hosts and for the total number of hosts that are starting up. Most of the limits and threshold variables in the script can be easily fine-tuned for balancing between performance and cost. Finally once the worker pool has been updated, the script writes any changes made to a log file.

5 Evaluation and future work

A set of tests were executed to evaluate the system. A batch of 30 randomly picked packages were built first with only LBH and then with both LBH and CBH. Each scenario was repeated three times for averaging. It should be noted that the computing power of local and cloud machines are not comparable. This measurement is only to give a rough idea how the cloudbursting works. On average, it took the LBH 153 minutes to build the batch, while with CBHs (small EC2 machines) it took 61 minutes. Using medium-sized high-CPU machines instead took 55 minutes.

The system uses an aggressive elasticity policy i.e. it gives explicitly a new worker for each job until the max number of workers is reached. This is a simple policy and it minimizes the build time but costs extra. However, when changes in the source code occur, it often reflects
into several packages through build-time dependencies. These packages have to be built in order, causing some jobs to wait for others to finish. Because of this, all packages cannot be always built in parallel. OBS has a feature to build the large bottleneck packages on specific, high-powered build hosts.

In total, 60 packages were built in the cloud in these tests. This cost $3.14 for disk and snapshot usage, $2.66 for network and $17.10 for computing itself. The traffic is a small portion of the total (with latest prices even lower) even though this setup did not cache any packages in the cloud. A cloud-side cache could be setup, shared between CBHs, to slightly lower the traffic cost and time spent. Smarter elasticity policies (e.g. do not launch a new instance if some job is about to finish) should also be implemented, but they require extra data from OBS, including estimates of build durations.

Main challenge with building in the cloud is that OBS is designed to be used in trusted server environments. First, if proprietary source code cannot be built in the cloud (as it has to be unencrypted during building), a mechanism is needed to restrict building of packages with certain license to local build hosts. Second, the source codes which are built can contain malicious code (originating from community-made packages) which is executed when setting up the build environment. To isolate the build environment, the OBS build host encapsulates it inside a virtual machine using KVM or Xen virtualization. However, as EC2 uses Xen as well (which does not support nested virtualization), support for OS-level virtualization such as LXC needs to be implemented in OBS.

A Virtual Infrastructure Manager (VIM) should be used for automating the management of the LBHs. In addition, VIMs like OpenStack and OpenNebula support multiple cloud providers allowing the customers to avoid vendor lock-in. The script itself also needs improvements in robustness, including graceful behaviour to disconnections. For more complex setups, some service managers exist, at least Claudia.

6 Conclusions

This paper presented how the infrastructure of a software build system can be distributed into domains that are geographically apart and owned by different organizations. Even though cloud services do provide cost-efficient flexibility, they have their security challenges in confidentiality, integrity and availability. Whether to keep a system in-house or move it to a cloud depends on many things, and it should be assessed whether cloud usage fits the use case. A hybrid model is a convenient way to keep business-critical data in-house, while buying extra computational power from the cloud.

Another aspect to consider is the cost, and it is not simple to estimate. With public infrastructure services, the share of traffic and storage is minimal compared to the cost of actual building. For constant needs, large enterprises that already have the means and knowledge are better off setting up a build cluster of their own. Cloud providers make profit by utilizing economies of scale, amassing servers together to save up in infrastructure and management costs.

Open Build Service is a powerful and flexible tool for building packages for multiple device architectures and target distributions. The main problem in the cloud usage of Open Build Service is that it has been designed for closed, trusted environments. Build hosts need to be classified based on their location to control where proprietary code is being built. Furthermore, OBS uses virtualization to inhibit attacks from inside of the build environments.
Due to cloud resources already being virtualized, support for lighter, OS-level virtualization is needed in OBS.

The cloudbursting build system setup in this work was a simplified proof-of-concept. With the implemented service manager, the system automatically accommodates to the number of waiting jobs, spawning new build hosts when needed and terminating them when not. While the machines from the cloud are not superior in performance, the fact that there is an seemingly endless pool of them makes the system good in flushing out a long queue of jobs that can be built in parallel. For large bottleneck packages, separate powerful build hosts need to be set up.

For production-use however, the system requires more work in terms of security, availability, manageability and performance. Infrastructure and service management software are needed in multi-tenant, multi-service and multi-provider environments. There are several competing alternatives for Virtual Infrastructure Management, but service managers seem to lack options. As services vary greatly, it is difficult to make a generic managed service API. Currently most services seem to rely on ad hoc service manager implementations.

References

4. Open Build Service (OBS). http://openbuildservice.org/